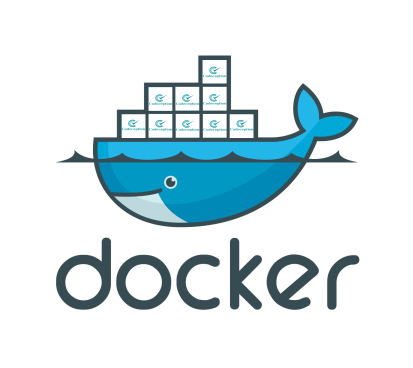
****

**What is Docker?**Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.

And importantly, Docker is open source. This means that anyone can contribute to Docker and extend it to meet their own needs if they need additional features that aren't available out of the box.

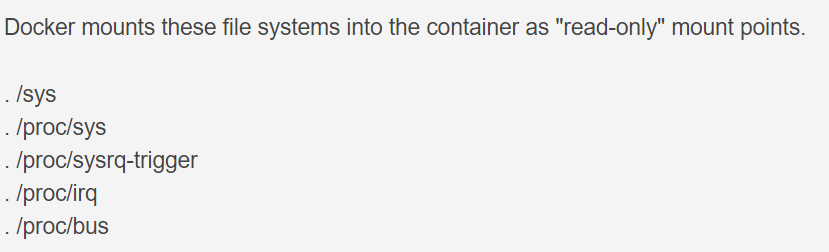
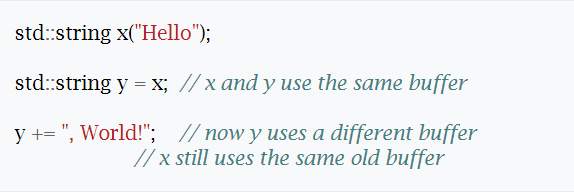
**Who is Docker for?**

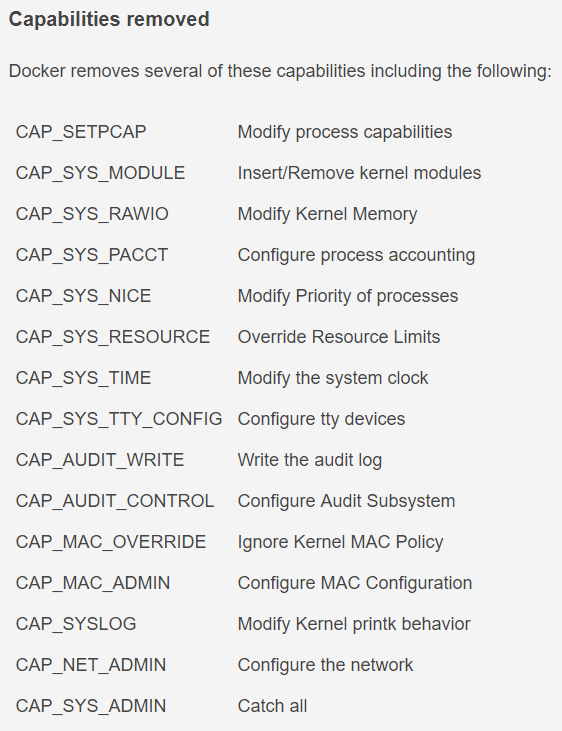
Docker is a tool that is designed to benefit both developers and system administrators, making it a part of many DevOps (developers + operations) toolchains. For developers, it means that they can focus on writing code without worrying about the system that it will ultimately be running on. It also allows them to get a head start by using one of thousands of programs already designed to run in a Docker container as a part of their application. For operations staff, Docker gives flexibility and potentially reduces the number of systems needed because of its small footprint and lower overhead

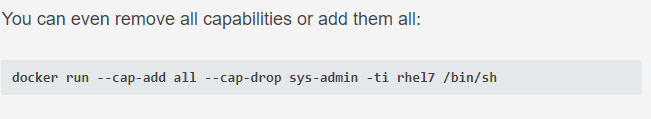
**Docker and security**

Docker brings security to applications running in a shared environment, but containers by themselves are not an alternative to taking proper security measures. With Docker we are using the layered security approach, which is "the practice of combining multiple mitigating security controls to protect resources and data.". Basically, we want to put in as **many security barriers** as possible to prevent a break out. If a privileged process can break out of one containment mechanism, we want to block them with the next. With Docker, we want to take advantage of as many security mechanisms of Linux as possible.Luckily, with Red Hat Enterprise Linux (RHEL) 7, we get a plethora of security features like :

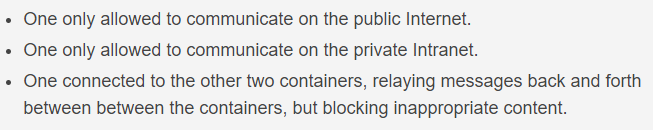
1. File System Protections :

* Read-only mount points : By mounting these file systems as read-only, privileged container processes cannot write to them. They cannot effect the host system
* Copy-on-write file systems : This prevents one container from seeing the changes of another container even if they wrote to the same file system image.
* Capabilities : Removing capabilities can cause applications to break, which means we have a balancing act in Docker between functionality, usability and security.





1. Namespaces :

* PID namespace : The PID namespace hides all processes that are running on a system except those that are running in your current container. If you can't see the other processes, it makes it harder to attack the process.
* Network namespace : The network namespace can be used to implement security. The admin can setup the network of a container with routing rules and iptables such that the processes within the container can only use certain networks.

1. Cgroups :

* One type of attack on a system could be described as a Denial Of Service. This is where a process or group of processes use all of the resources on a system, preventing other processes from executing. cgroups can be used to mitigate this by controlling the amount of resources any Docker container can use. For example the CPU cgroup can be setup such that an administrator can still login to a system where a Docker container is trying to dominate the CPU and kill it. Docker blocks the processes from creating and using device nodes that could be used to attack the host.

1. AppArmor :

* Apparmor is available for Docker containers on systems that support it

1. SELinux :

* SELinux implements a Mandatory Access Control system. This means the owners of an object have no control or discretion over the access to an object. The kernel enforces Mandatory Access Controls.

1. Type enforcement :

* Type Enforcement protects the host from the processes within the container

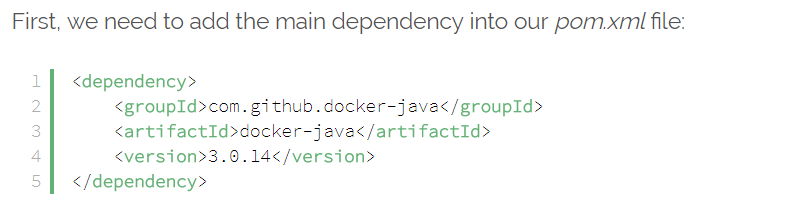


1. Multi Category Security enforcement

* Multi Category Security is based on Multi Level Security (MLS). MCS takes advantage of the last component of the SELinux label the MLS Field. MCS enforcement protects containers from each other.

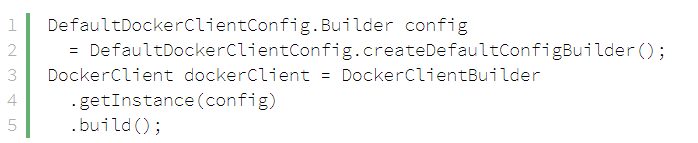
References : https://opensource.com/business/14/9/security-for-docker

**Docker API :**

1. **Maven Dependency**
2. **Using the Docker Client**

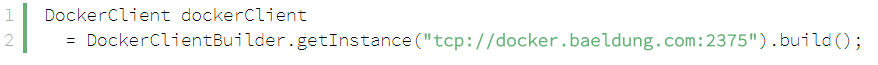
DockerClient is where we can establish a connection between a Docker engine/daemon and our application. By default, the Docker daemon can only be accessible at the unix:///var/run/docker.sock file. We can locally communicate with the Docker engine listening on the Unix socket unless otherwise configured. Here, we apply to the DockerClientBuilder class to create a connection by accepting the default settings:

Similarly, we can open a connection in two steps:



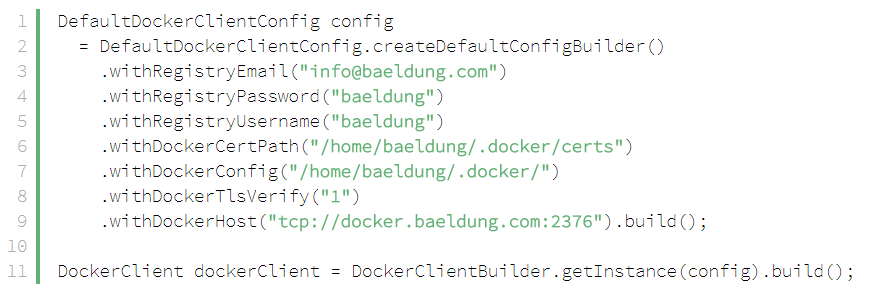
Since engines could rely on other characteristics, the client is also configurable with different conditions.

For example, the builder accepts a server URL, that is, **we can update the connection value if the engine is available on port 2375**:

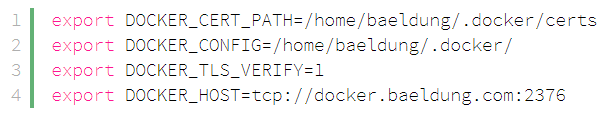
****

Note that we need to prepend the connection string with **unix:// or tcp:// depending on the connection type.**

we can end up with a more advanced configuration using the DefaultDockerClientConfig class:



Another choice unless we configure the engine’s settings in the source code is to set corresponding environment variables so that we can only consider the default instantiation of DockerClient in the project:



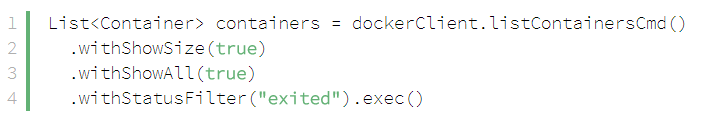
1. **Container Management**

The API allows us a variety of choices about container management. Let’s look at each one of them.

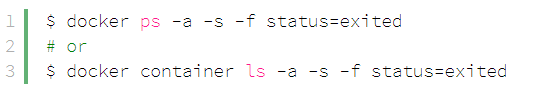
* 1. **List Containers**

Now that we have an established connection, we can list all the running containers located on the Docker host: ****

Provided that showing the running containers doesn’t appeal the need, we can make use of the offered options to query containers.

In this case, we display containers with the “exited” status: 

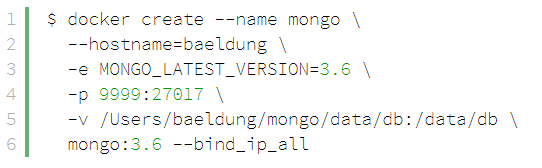
It’s an equivalent of:



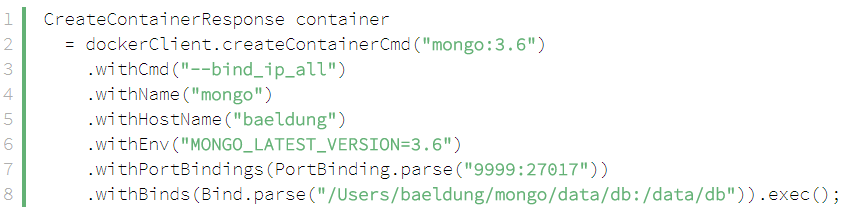
* 1. **Create a Container**

Creating a container is served with the createContainerCmd method. We can declare more complex declaration **using** **the available methods starting with the “with” prefix**.

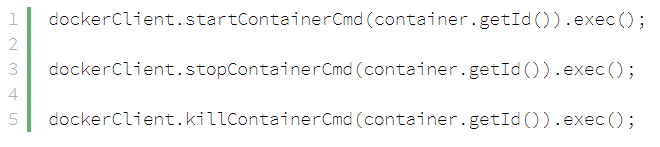
Let’s assume that we have a docker create command defining a host-dependent MongoDB container listening internally on port 27017



We’re able to bootstrap the same container along with its configurations programmatically:

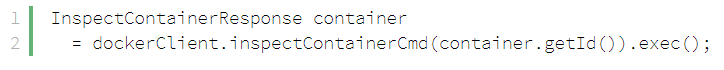


* 1. **Start, Stop, and Kill a Container**

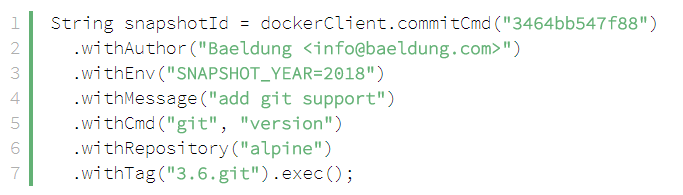
Once we create the container, we can start, stop and kill it by name or id respectively:

### ****Inspect a Container****

The inspectContainerCmd method takes a String argument which indicates the name or id of a container. Using this method, we can observe the metadata of a container directly:



* 1. **Snapshot a Containe**

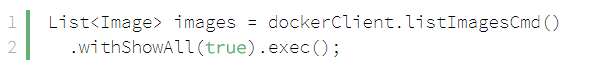
Similar to the docker commit command, we can create a new image using the commitCmd method.

1. **Image Management**

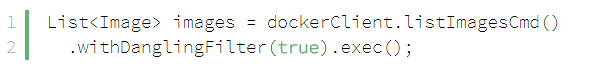
There are a few applicable commands we are given to manage image operations**.**

* 1. **List Images**

To list all the available images including dangling images on the Docker host, we need to apply to the listImagesCmd method:

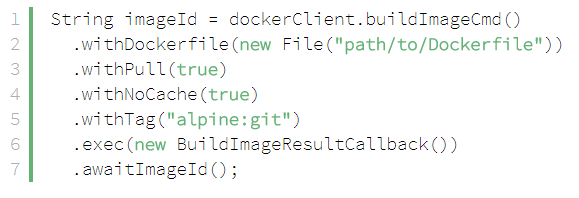
to see the intermediate images, we need to request it explicitly:

If only displaying the dangling images is the case, the withDanglingFilter method must be considered:



**4.2 Build an Image**

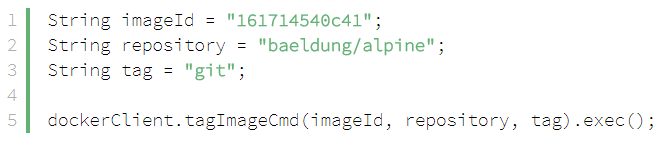
the way of building an image using the API. The **buildImageCmd method builds Docker images from a Dockerfile.**

****

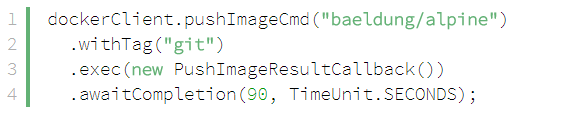
### ****Inspect an Image****

### We can inspect the low-level information about an image thanks to the inspectImageCmd method:

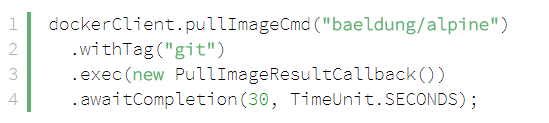
### ****Tag an Image****

Adding a tag to our image is quite simple using the docker tag command, so the API is no exception. 

* 1. **Push an Image**

Before sending out an image to a registry service, the docker client must be configured to cooperate with the service because working with registries need to be authenticated in advance

### ****Pull an Image****

To download images from registry services, we make use of the pullImageCmd method. In addition, if the image being pulled from a private registry. 

* 1. **Remove an Image**

Another simple function among the rest is the removeImageCmd method. We can remove an image with its short or long ID: 

* 1. **Search in Registry**

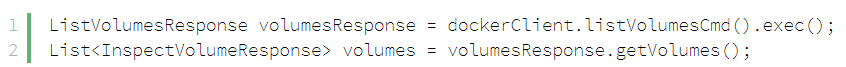
To search an image from Docker Hub, the client comes with the searchImagesCmd method taking a String value which indicates a term.



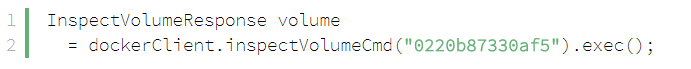
1. **Volume Management**

Java projects need to interact with Docker for volumes, So **we look at the fundamental techniques of volumes provided by the Docker Java API**

### ****List Volumes****

All of the available volumes including named and unnamed are listed with: 

* 1. **Inspect a Volume**

We inspect the volume by specifying its short id: 

* 1. **Create a Volume**

****

* 1. **Remove a Volume**

We can intuitively delete a volume from the Docker host using the removeVolumeCmd method. **What is important to note that we cannot delete a volume if it is in use from a container. **

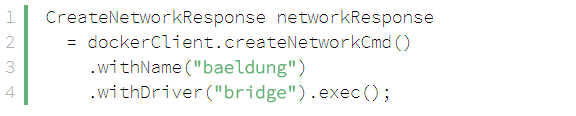
1. **Network Management**

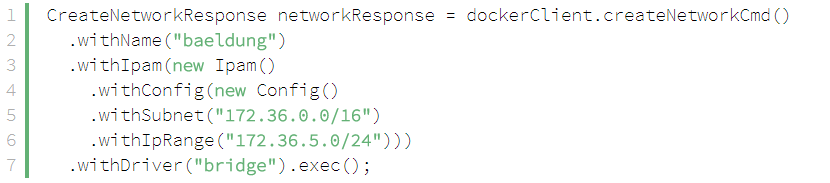
### ****List Networks****



* 1. **Create a Network**

The equivalent of the docker network create command is conducted with the createNetworkCmd method.



Furthermore, creating a network unit with the default settings doesn’t solve the problem, we can apply for other helper methods to construct an advanced network. Thus, to **override the default subnetwork with a custom value: **

* 1. **Inspect a Network**

****

* 1. **Remove a Network**

****

References : <https://docs.docker.com/engine/api/v1.24/#3-endpoints>